Memory Management: Headers

Where are we? We are talking about memory, and we have already covered argument passing, scoping, pointers and arrays. We have also talked about how Java has a garbage collector but C++ does not, so in C++ we have to manage our own dynamic memory. Now we need techniques for actually managing dynamic memory.

Remember: local variables are on the stack, and their lifetime is limited to the function/method in which they are created. Dynamic data is on the heap; it is created via the new operator and lives until it is deleted (via the delete operator).

Note that dynamic data never has a name. It’s data, but it isn’t a variable. Only local variables on the stack get names. Instead, dynamic data is data that is allocated on the heap and manipulated via pointers. The pointers are often named variables, but the pointers are not the same thing as the data they point to. The pointer is nothing less and nothing more than a typed virtual memory address.

If you allocate memory on the heap and never delete it, your program has a memory leak. So we need strategies for memory management. Let’s start with the simplest one, which I call memory management strategy #0:

- Only use local variables
  - Never use the new or delete operators.
  - Never explicitly use pointers
  - Never return a pointer or a reference to a local variable.

If you never use the heap, you won’t leak memory. More specifically, we already discussed how local variables enter and leave scope, and that for objects their constructors are called when they enter scope and their destructors are called when they leave scope. Since local variables leave scope when a method returns, their destructors will be called and their memory will be reclaimed. No memory will be leaked.

Unfortunately, this strategy doesn’t work for dynamic data, i.e. data whose size is not known until run-time. Local variables aren’t allowed to be dynamic. This is because they are stored on the stack, and the compiler needs to know how big the stack frame will be at compile time. Therefore, only data whose size is known at compile-time can be kept in local variables.

But what about the vector class? Isn’t it dynamic? Can’t I have a local variable that is a vector? Yes, and it is the canonical example of the header class scheme I call memory management strategy #1, and that I want you all to learn for hiding dynamic memory.

Under the hood, vectors are implemented as arrays on the heap. The vector class holds a pointer to this array, along with other bookkeeping information like the size of the array.

This is why I emphasized that dynamic memory isn’t named. The pointer to the memory is usually the variable that is named, and it is a local variable.

This is how vectors work. The vector class is like an abstract pointer – a header, really – that holds the pointer to the heap and auxiliary information.
Vectors can therefore be local variables that hide the fact that the underlying data is on the heap.

Let’s see if we can figure this out in more detail. If we are asked to implement a class for a vector of integers from scratch, how should we do it?

We start by creating a header class that holds a pointer to the data on the heap and the number of elements stored on the heap:

```cpp
class int_vector {
public:
    int_vector(int sz = 0);
    ~int_vector();

private:
    int* data;
    int size;
};
```

Note the `~` syntax. It is the logical not operator, and is used to declare destructors. We have talked about destructors in lecture before, but this may be the first time you have seen one. (Well, there was one in the Complex class, but that was a while ago.)

How do we implement the `int_vector` constructor? The idea is that we want to make the header class a local (stack) variable, but have it point to dynamic data on the heap. The only primitive dynamic structure we have is the array, and the only operator for allocating heap memory is `new`.

So we write the constructor (in an `intvector.cpp` file) as:

```cpp
int_vector::int_vector(int sz) : size(sz) {
    if (size == 0) data = NULL;
    else data = new int[size];
}
```

This means that when we create a local variable of type `int_vector`, the constructor will be called automatically and will allocate the appropriate data on the heap. Now we define the destructor to do the opposite – to get rid of the data on the heap.

```cpp
int_vector::~int_vector() {
    delete [] data;
    data = NULL;
}
```

The general idea here is to make the header class `int_vector` act like a local variable, even though it actually has to stash data on the heap. When you make an `int_vector` variable, its constructor allocates space dynamically on the heap, and when the `int_vector` drops out of scope, its destructor deletes the memory automatically. This way you, the programmer using the `int_vector` class, never sees the pointer to the data on the heap. The memory is allocated and then deleted automatically.

Before we go on, there are some notes about the destructor example above. One is the line `delete [] data;`, and specifically the empty square brackets `[]`. Remember that data is a pointer to int, and that a pointer is nothing more than a typed memory address. If we typed `delete data;` we would be
telling the compiler to delete what data points to, and data points to an int. A single int. So it would delete one integer, and leave the rest of the array. And we would have a memory leak.

The empty square brackets tell the compiler that data is pointing to the first integer in an array, and that the whole array should be deleted. (The OS knows what size blocks it allocated, so it knows how many integers to delete.)

Also, the delete operator deletes the memory that a pointer points to, but it doesn’t change the value of the pointer. The pointer holds the same memory address that it did before the call to delete. Unfortunately, after the call to delete the memory is no longer allocated and therefore following the pointer will yield undefined values. Therefore it is good style to set the pointer to NULL after the call to delete, to make sure you do not accidently use it later. (Also, calling delete on a NULL pointer is a no-op.)

So now we have a header class that allocates dynamic memory on the heap when it enters scope and deletes that memory when it leaves scope. Even better, the pointer itself is protected, so the application program can’t mess with it. Unfortunately, our class is useless because we can’t access any of the data in the int_vector! So we need to add some access functions…

class int_vector {
public:
    int_vector(int sz = 0);
    ~int_vector();
    int at(unsigned int index) const;
    int& at(unsigned int index);
private:
    int* data;
    int size;
}

And then we can implement them in int_vector.cpp:

int int_vector::at(unsigned int index) const
{
    if (index < size) return data[index];
    else throw std::exception();
}

int& int_vector::at(unsigned int index)
{
    if (index < size) return data[index];
    else throw std::exception();
}

The \texttt{at} method is used to get data out of the \texttt{int\_vector}. It is similar to the \texttt{[i]} notation for arrays, but slightly different. Notice above that the \texttt{at} methods check the range of their argument – if index is too big, they throw an error. (See your textbook for notes on throwing errors. It’s essentially the same in \texttt{C++} as it is in \texttt{Java}.) The standard \texttt{vector} and \texttt{string} classes you have been using in \texttt{C++} have \texttt{at} methods too, and they check that the index is in bounds. This is really useful and prevents a lot of errors. The brackets don’t check the size of the index, they just add it to the base pointer as an offset. If that goes off into unallocated memory, well, that’s the programmer’s problem (i.e. a bug).
As a best practice, I recommend always using \textit{at} methods instead of brackets until your code has been thoroughly debugged. Then, if it proves necessary for efficiency, you can replace the most common calls to \textit{at} with square brackets.

Why are there two versions of \textit{at()} method? Note that although the methods have the same name (\textit{at}) and take the same explicit arguments (a single unsigned int), they are different because their hidden arguments are different. The first version takes a \texttt{const int\_vector\&} hidden argument, while the second takes an \texttt{int\_vector\&} hidden argument. Therefore the compiler selects which version to call based on whether the hidden argument is constant or not.

But why have two versions? You need the non-constant version of \textit{at} so that you can set values by using the reference on the left hand side of an assignment operator, for example:

\begin{verbatim}
int\_vector ivec(5);
ivec.at(0) = 0;
\end{verbatim}

This is legal, because you can store into a reference. At the same time, however, you would like to be able to pass a constant reference to an \texttt{int\_vector} to a method so that we can avoid copying it while telling other programmers (and the compiler) that the method isn’t going to side-effect the \texttt{int\_vector}. Therefore we need a version of \textit{at} that operates on non-constant \texttt{int\_vector}s. That is the first version above, and they are different because it returns integer values, not references to integers.

Together, the constructors, destructors and \textit{at} methods allow us to create a vector whose size was unknown at compile time (and passed into the constructor as a parameter). We create the header as a local (stack) variable, and when it is created the heap memory is allocated, and when it falls out of scope the memory is deleted by the destructor. In between, we can set and read the values in the array.

But what if we want it to be more dynamic; what if we want the vector to be able to grow after its been created? To do this, we add a \texttt{push\_back} method.

\begin{verbatim}
class int\_vector {
public:
   int\_vector(int sz = 0);
~int\_vector();
   int at(unsigned int index) const;
   int& at(unsigned int index);
   void push\_back(int value);

private:
   int* data;
   int size;
}
\end{verbatim}

The idea is that \texttt{push\_back} makes the vector larger. But how do we do this? As follows:

\begin{verbatim}
void int\_vector::push\_back(int value) {
   int* temp = data;
   data = new int[size+1];
   for(int index=0; index < size; index++) {
      data[index] = temp[index];
   }
   for(int index=size; index < size+1; index++)
      data[index] = value;
}
\end{verbatim}
The \texttt{push\_back} method allocates a new vector on the heap, in this implementation one larger than the previous array, and then copies the data from the old array to the new array. It then stores the new value (from the parameter) at the end of the array, increments \texttt{size}, and deletes the original.

It is not uncommon to have a method like \texttt{push\_back} that dynamically changes the amount of memory in a data structure. We will talk more about \texttt{push\_back} and efficiency next lecture, but at the moment we focus on its role in memory management. \texttt{Push\_back} works because it (1) allocates a new, larger memory block on the heap, (2) copies data from the old memory block to the new one, and then (3) deletes the old memory block. So all the pointer manipulation is hidden from the application programmer.

Moreover, memory is never leaked. The trick to managing memory is to make sure that every \texttt{new} is balanced by a \texttt{delete}, and we have done that. The \texttt{int\_vector} is a header class on the stack. Therefore it will be created by a constructor which calls \texttt{new}, and when it falls out of scope its destructor calls the matching \texttt{delete}. Every \texttt{new} is matched by a \texttt{delete}. If it has to be resized, \texttt{push\_back} may be called, but this introduces one more call to \texttt{new} and one more call to \texttt{delete}, so everything remains balanced.

Well, almost. We still have some problems. Consider the following two functions:

\begin{verbatim}
int Foo(int_vector a)
{
    return a.at(0) * a.at(1);
}

int Bar(int value)
{
    int_vector b(2);
    b.at(0) = 1;
    b.at(1) = 2;
    return Foo(b) * b.at(1);
}
\end{verbatim}

What does this code do? It crashes! Why? Because the \texttt{int\_vector} parameter to \texttt{Foo} was passed by value, not by reference. Now this should be OK – in C++, we should be able to pass any parameter type by value. But the default way of copying an object is by copying every field of the object. In this case, that means copying every field of the \texttt{int\_vector}, including \texttt{data}, which is a pointer. Therefore the \texttt{int\_vector} \texttt{a} inside \texttt{Foo} points to the same array on the heap as the \texttt{int\_vector} \texttt{b} inside \texttt{Bar}. Having two pointers to a single block of data is called aliasing, and in this case it is a bug. Because when \texttt{Foo} returns, the \texttt{int\_vector} destructor is called on \texttt{a}. This deletes the array on the heap. But when \texttt{Foo} returns, the \texttt{int\_vector} \texttt{b} still has a pointer to the array that was just deleted! When \texttt{b} tries to access this pointer, the system (most likely) crashes.

The problem is that when the \texttt{int\_vector} is passed to \texttt{Foo}, a new instance of the \texttt{int\_vector} header is created without calling \texttt{new}. When this copy falls out of scope, its destructor introduces one too many calls to \texttt{delete}. What we need is another way to make a copy of \texttt{int\_vector}. 

When an object is passed by value, it is copied using a *copy constructor*. A copy constructor is a constructor that takes a constant reference to an object of the same type that is being initialized. Every object gets a default copy constructor that copies the fields of the parameter to the fields of the object being initialized. This is what we want – unless the object has a pointer. Then we need to do something else. Then we need to copy the data that the pointer points to.

Let’s define and implement a copy constructor for `int_vector`:

```
class int_vector {
public:
   int_vector(int sz = 0);
   int_vector(const int_vector& src);
   ~int_vector();
   int at(unsigned int index) const;
   int& at(unsigned int index);
   void push_back(int value);
private:
   int* data;
   int size;
};

int_vector::int_vector(const int_vector& src) : size(src.size) {
   data = new int[size];
   for(int i=0; i < size; i++) {
      data[i] = src.data[i];
   }
}
```

This is what is known as a deep copy constructor. It allocates new memory, and then copies the data from the source’s heap memory to the new heap memory it allocates. This way there is no aliasing. Both instances have their own pointers, pointing to their own arrays on the heap. When one is deleted, the other is still OK.

Now we can treat `int_vector`s like dynamic local variables, without memory leaks. Every instance allocates and deletes its own memory, even if the instance is created as pass-by-value.

Well, one more thing. What about assignment? What if the application programmer writes:

```
int_vector a;
int_vector b;
...
```

```
a = b;
```

Now what happens? What should happen is that (1) the old memory in `a` is deleted, (2) new memory is allocated for `a`, and (3) the data pointed to by `b` is copied to the memory block of `a`. We make this happen like this:

```
class int_vector {
public:
   int_vector(int sz = 0);
   int_vector(const int_vector& src);
   ~int_vector();
   int_vector& operator = (const int_vector& src);
};
```
int at(unsigned int index) const;
int& at(unsigned int index);
void push_back(int value);

private:
    int* data;
    int size;
};

int_vector& int_vector::operator = (const int_vector& src) {
    delete [] data;
    size = src.size;
    data = new int[size];
    for(int i=0; i < size; i++) {
        data[i] = src.data[i];
    }
    return *this;
}

So in summary, memory management strategy #1 is to use header classes. Header classes are classes that manage dynamic memory on the heap through their constructors, destructors and assignment operators. They sometimes have other methods that manage memory as well (e.g. push_back). All access to the dynamic data is through methods of the header class, and the pointer to the heap is kept protected. In this way, the dynamic data and its access methods are encapsulated in the header class.

Header classes allow programmers to manage the heap data systematically. Header class objects can be used as local variables. As local variables, constructors are called when they are created (including copy constructors if they are created via pass-by-value) and destructors are called when they fall out of scope. The constructors all call new, and the destructors call the matching delete. No memory is leaked. Assignment introduces both a new and delete, so the calls still match up. Finally, another other method that allocates memory calls both new (to allocate new memory) and delete (to get rid of the old memory), guaranteeing that all allocated memory is deleted once and only once.